Using MMX™ Instructions to Convert 24-Bit True Color Data to 16-Bit High Color

Information for Developers and ISVs

From Intel® Developer Services
www.intel.com/IDS

March 1996

Information in this document is provided in connection with Intel® products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Note: Please be aware that the software programming article below is posted as a public service and may no longer be supported by Intel.

Copyright © Intel Corporation 2004
* Other names and brands may be claimed as the property of others.

Using MMX™ Instructions to Convert 24-Bit True Color Data to 16-Bit High Color

March 1996

CONTENTS

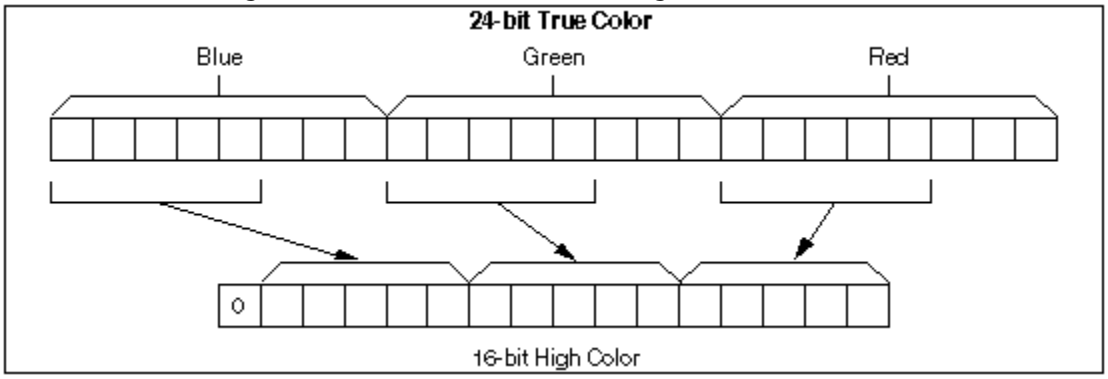
1.0. INTRODUCTION

The Intel Architecture (IA) media extensions include single-instruction, multi-data (SIMD) instructions. This application note presents examples of code that exploit these instructions. Two RGB24to16 functions are examined that use the new MMX instructions PSRLD, PXOR, PACKSS, and PMADD to complete the conversion. The performance improvement relative to traditional IA code can be attributed primarily to the much faster shift instructions. Whereas the IA shift instruction (SHIFT) takes four cycles on a Pentium® processor, the MMX shift instruction (PSHIFT) takes only one cycle. Also, the new PMADD instruction combines a multiply-and-add function which has a throughput of one and a latency of three cycles.

2.0. RGB32TO16 FUNCTIONS

Many applications provide RGB data with the assumption that the video display will use 24-bit true color data. This is especially true of three-dimensional (3D) applications that perform calculations to generate true color values for the pixels to be displayed. Each byte contains one byte of Red, Green, or Blue data. There continues to be, however, an abundance of video displays that require 16-bit high color data. Visually, the conversion to RGB555 is straightforward. There are other pixel formats, such as RGB565, to which the techniques of this paper can be easily applied.

Figure 1. 24-bit True Color to 16-bit High Color Conversion



In the sections that follow, two conversion methods will be discussed. The first method, which will be referred to as the Mask-Shift-Or method, is the algorithm that is traditionally used to complete the conversion. This method uses very few MMX registers, making it possible to interleave the processing of pixels to efficiently use the processor's clock cycles.

The second method, which will be referred to as the PMADD method, makes use of the multiply-and-add instruction to shift the bits into their appropriate positions. This method uses the multiply instruction to shift the data and the add instruction to combine the bits together. Even though the PMADD instruction has a latency of three cycles, it is a pipelined instruction that can be executed in either the U or V pipe, allowing other instructions to be paired with it. As illustrated in Section 2.2, this method takes advantage of this and processes the pixels even more efficiently than the Mask-Shift-Or algorithm.

Mask-Shift-Or Algorithm

The Mask-Shift-Or algorithm takes each 24-bit true color element, stored in the three least significant bytes of a double word, masks each 8-bit color, shifts it right three bits, and ORs the result into a register. Using the 64-bit MMX registers and instructions, two pixels can be processed at the same time.

Example 1. Mask-Shift-Or Basic Algorithm

```
1       movq    mm0, [eax]              ;get two 24-bit pixels
2       movq    mm1, mm0        ;save the original data
3       pand    mm0, BLUES              ;mask out all but the 5 MSB blue bits
4       psrld   mm0, 3                  ;shift blue bits to bits 0-4
5       movq    mm2, mm1        ;save the original data again
6       pand    mm1, GREENS             ;mask out all but the 5 MSB green bits
7       psrld   mm1, 6                  ;shift green bits to bits 5-9
8       por     mm0, mm1        ;or in the green bits with the blue bits
9       pand    mm2, REDS               ;mask out all but the 5 MSB red bits
10      psrld   mm2, 9                  ;shift red bits to bits 10-14
11      por     mm0, mm2        ;or in the red bits
```

```
                                                ;mm0 now contains 2 16-bit color elements
                                                ;in the low word of each DWORD
        12      packssdw mm0, ZERO              ;pack the 2 16-bit elements into low DWORD
```

The code example shown in Example 1 does little to take advantage of instruction pairing and uses eleven cycles to process two pixels. It also leaves the upper DWORD of the result register unused. By processing another two pixels and pairing the instructions to make the best use of the U and V pipes, four pixels can be processed in thirteen cycles or 3.25 cycles per pixel. The comments indicate which pair of pixels is being affected and the instruction step as illustrated in Example 1.

Example 2. Mask-Shift-Or Paired Algorithm

```
1       movq            mm0, [eax]      ;(A1)   get two 24-bit pixels
2       movq            mm3, 8[eax]     ;(B1)   get next two 24-bit pixels
3       movq            mm1, mm0 ;(A2)  save the original first two
4       pand            mm0, BLUES      ;(A3)   mask out all but the 5 MSB blue bits
5       movq            mm4, mm3 ;(B2)  save the original second two
6       pand            mm3, BLUES      ;(B3)   mask out all but the 5 MSB blue bits
7       psrld           mm0, 3          ;(A4)   shift blue bits to bits 0-4
8       psrld           mm3, 3          ;(B4)   shift blue bits to bits 0-4
9       movq            mm2, mm1 ;(A5)  save the original data again
10      pand            mm1, GREENS     ;(A6)   mask out all but the 5 MSB green bits
11      movq            mm6, mm4 ;(B5)  save the original data again
12      pand            mm4, GREENS     ;(B6)   mask out all but the 5 MSB green bits
13      psrld           mm1, 6          ;(A7)   shift green bits to bits 5-9
14      por             mm0, mm1 ;(A8)  or in the green bits with the
                                        ;       blue bits
15      psrld           mm4, 6          ;(B7)   shift green bits to bits 5-9
16      pand            mm2, REDS       ;(A9)   mask out all but the 5 MSB red bits
17      por             mm3, mm4 ;(B8)  or in the green bits with the
                                        ;       blue bits
18      pand            mm6, REDS       ;(B9)   mask out all but the 5 MSB red bits
19      psrld           mm2, 9          ;(A10)  shift red bits to bits 10-14
20      por             mm0, mm2 ;(A11) or in the red bits
21      psrld           mm6, 9          ;(B10)  shift red bits to bits 10-14
22      por             mm3, mm6 ;(B11) or in the red bits
23      packssdw mm0, mm3 ;(AB12) pack the 4 16-bit pixels into
                                        ;       one qword
```

It would have been more straightforward to simply dedicate the U pipe to the A pair of pixels and the V pipe to the B pair. However, memory accesses can only be done in the U pipe making it necessary to mix the use of the two pipes between the two pairs of pixels.

Two instructions still remain unpaired at the end of the paired algorithm. However, because each pair of pixels is converted using only three MMX registers, four more pixels can be processed if the instructions are interleaved allowing more pairing to be done. This algorithm is completed in twenty-four cycles for each set of eight pixels or three cycles per pixel.

Example 3. Mask-Shift-Or Interleaved Algorithm

```
1       movq            mm0, [eax]      ;(A1)   get two 24-bit color elements
2       movq            mm3, 16[eax]    ;(C1)   get third two 24-bit pixels
3       movq            mm1, mm0 ;(A2)  save the original first two
4       pand            mm0, BLUES      ;(A3)   mask out all but the 5 MSB blue bits
5       movq            mm4, mm3 ;(C2)  save the original second two
6       pand            mm3, BLUES      ;(C3)   mask out all but the 5 MSB blue bits
7       psrld           mm0, 3          ;(A4)   shift blue bits to bits 0-4
```

```
8       psrld           mm3, 3          ;(C4)   shift blue bits to bits 0-4
9       movq            mm2, mm1 ;(A5)  save the original data again
10      pand            mm1, GREENS     ;(A6)   mask out all but the 5 MSB
                                        ;       green bits
11      movq            mm6, mm4 ;(C5)  save the original data again
12      pand            mm4, GREENS     ;(C6)   mask out all but the 5 MSB
                                        ;       green bits
13      psrld           mm1, 6          ;(A7)   shift green bits to bits 5-9
14      por             mm0, mm1 ;(A8)  or in the green bits with the blue
15      psrld           mm4, 6          ;(C7)   shift green bits to bits 5-9
16      pand            mm2, REDS       ;(A9)   mask out all but the 5 MSB red bits
17      por             mm3, mm4 ;(C8)  or in the green bits with the blue
18      pand            mm6, REDS       ;(C9)   mask out all but the 5 MSB red bits
19      psrld           mm2, 9          ;(A10)  shift red bits to bits 10-14
20      por             mm0, mm2 ;(A11) or in the red bits
21      psrld           mm6, 9          ;(C10)  shift red bits to bits 10-14
22      movq            mm1, 24[eax]    ;(D1)   get fourth two 24-bit pixels
23      por             mm6, mm3 ;(C11) or in the red bits
24      movq            mm3, 8[eax]     ;(B1)   get second two 24-bit pixels
25      movq            mm2, mm1 ;(D2)  save the original data
26      pand            mm1, BLUES      ;(D3)   mask out all but the 5 MSB blue bits
27      movq            mm4, mm3 ;(B2)  save the original data
28      pand            mm3, BLUES      ;(B3)   mask out all but the 5 MSB blue bits
29      psrld           mm1, 3          ;(D4)   shift blue bits to bits 0-4
30      psrld           mm3, 3          ;(B4)   shift blue bits to bits 0-4
31      movq            mm5, mm4 ;(B5)  save the original data again
32      pand            mm4, GREENS     ;(B6)   mask out all but the 5 MSB
                                        ;       green bits
33      movq            mm7, mm2 ;(D5)  save the original data again
34      pand            mm2, GREENS     ;(D6)   mask out all but the 5 MSB
                                        ;       green bits
35      psrld           mm4, 6          ;(B7)   shift green bits to bits 5-9
36      por             mm3, mm4 ;(B8)  or in the green bits with the blue
37      psrld           mm2, 6          ;(D7)   shift green bits to bits 5-9
38      pand            mm5, REDS       ;(B9)   mask out all but the 5 MSB red bits
39      por             mm1, mm2 ;(D8)  or in the green bits with the blue
40      pand            mm7, REDS       ;(D9)   mask out all but the 5 MSB red bits
41      psrld           mm5, 9          ;(B10)  shift red bits to bits 10-14
42      por             mm3, mm5 ;(B11) or in the red bits
43      psrld           mm7, 9          ;(D10)  shift red bits to bits 10-14
44      packssdw mm0, mm3 ;(AB12) pack result 1 and 2 into one qword
45      por             mm7, mm1 ;(D11) or in the red bits
46      packssdw mm6, mm7 ;(CD12) pack result 3 and 4 into one qword
```

All that remains is to add instructions to write the results out to memory and to loop through the source. See Appendix A for the full source code for the Mask-Shift-Or algorithm. The instructions to calculate the value of the loop counter were written for this application and may need to be changed to meet the needs of the user's application. Also note that the source data is processed from the end of the source array instead of the beginning in order to eliminate the need for a compare instruction before the jump to the beginning of the loop.

Multiply-Add Algorithm

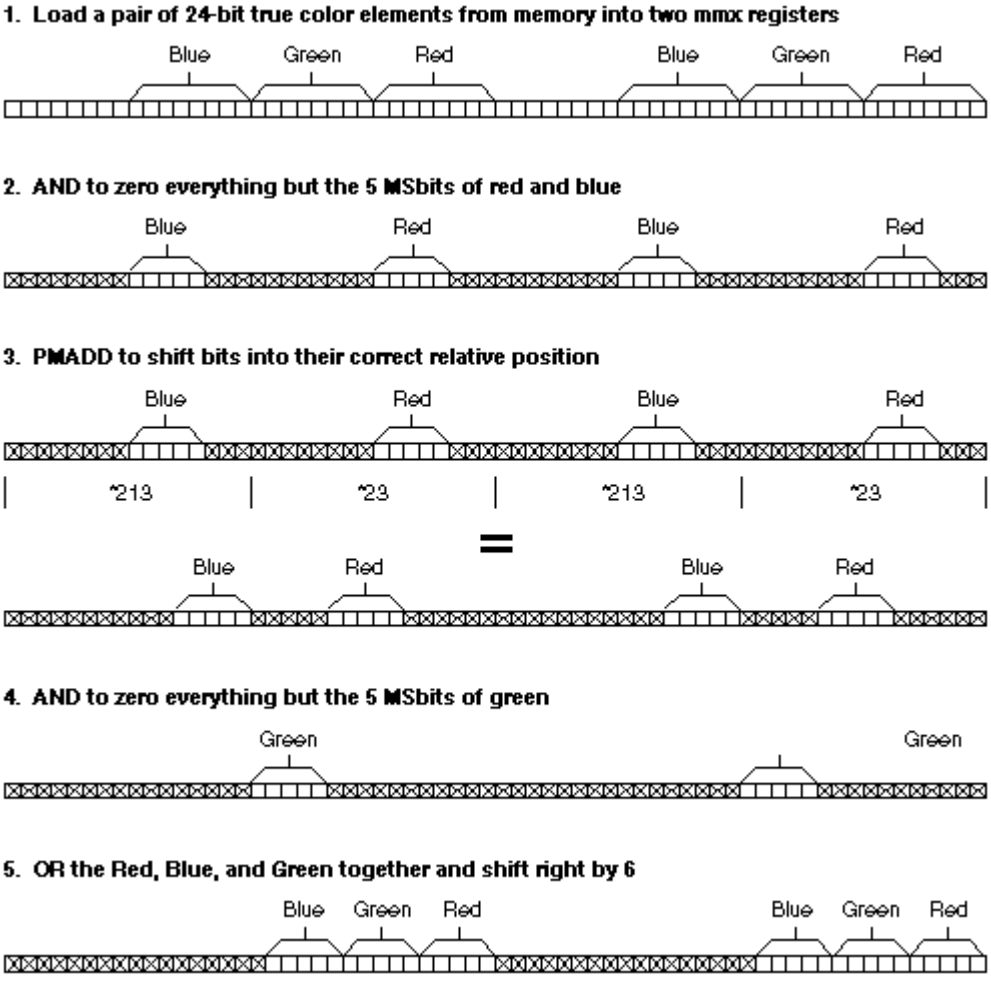
The Multiply-Add algorithm is less straightforward but takes advantage of the PMADD instruction to shift and combine the bits into their final position. It also uses the 64-bit MMX registers and instructions to convert two 24-bit true color elements to 16-bit high color at the same time.

The PMADD instruction is capable of multiplying each word of a quaDWORD by a unique word value. It then adds the two low-order words and puts the result in the lower doubleword of the result register. Likewise, the two high-order words are added with the result written into the high order double word of the result register. This instruction only supports word multiplication's with the result of the PMADD instruction written as double words.

The first trap to avoid is the assumption that it is necessary to unpack the three bytes of color information into three words in order to use the PMADD instruction. By initially ignoring the green byte, the red and blue bytes can be operated on as words. By carefully choosing the multiplication factor used for each of these colors, the red and blue bytes can be shifted into their relative final position and simply OR-ORed with the green bits.

The flow of the basic algorithm is depicted in Figure 2.

Figure 2. PMADD Basic Algorithm



Example 4. PMADD Basic Algorithm

```
1       movq            mm0, [eax]              ;get two 24-bit pixels
2       movq            mm1, mm0        ;save the original data
```

```
3       pand            mm0, REDBLUE            ;mask out all but the 5MSBits of red and
blue
4       pmaddwd         mm0, MULFACT            ;multiply each word by
                                                ;2**13, 2**3, 2**13, 2**3 and add results
5       pand            mm1, GREEN              ;mask out all but the 5MSBits of green
6       por             mm0, mm1        ;combine the red, green, and blue bits
7       psrld           mm0, 6                  ;shift to final position
```

Example 4 shows the implementation of the basic algorithm using MMX instructions. As in the Mask-Shift-Or method, processing only one pair of pixels does little to take advantage of the processor's ability to pair instructions. In fact, none of the instructions in the basic algorithm pair. This algorithm is very efficient, however, in its use of registers allowing for more pixels to be processed within the same loop.

By processing another six pixels and pairing the instructions to make the best use of the U and V pipes, eight pixels can be processed in nineteen cycles or 2.375 cycles per pixel. The comments indicate which pair of pixels is being affected and the instruction step as illustrated in Example 4. Unlike the PMADD instructions in lines 7 and 8, the PMADD instruction in lines 21 and 24 cause pipeline stalls because the results need to be used before the three clock cycles that the multiplier needs have been completed. Also, because mm5 is used as the destination in the last three instructions, these instructions do not pair.

Example 5. PMADD Paired Algorithm

```
;assume mm6 holds mask for 5MSbits of green
;assume mm7 holds the multiplication factor 2^13,2^3,2^13,2^3

1       movq            mm2, 8[eax*4]           ;(B1)   get second two 24-bit pixels
2       movq            mm0, [eax]              ;(A1)   get first two 24-bit pixels
3       movq            mm3, mm2        ;(B2)   save the original data
4       pand            mm3, REDBLUE            ;(B3)   mask for 5MSbits of red and blue
5       movq            mm1, mm0        ;(A2)   save the original data
6       pand            mm1, REDBLUE            ;(A3)   mask for 5Msbits of red and blue
7       pmaddwd         mm3, mm7        ;(B4)   use multiply to shift and or
                                                ;       can't use mm3 for 3 cycles!
8       pmaddwd         mm1, mm7        ;(A4)   but we can do another multiply
9       pand            mm2, mm6        ;(B5)   mask for 5MSbits of green
10      movq            mm4, 24[eax]            ;(D1)   get fourth two 24-bit pixels
11      pand            mm0, mm6        ;(A5)   mask for 5MSbits of green
12      movq            mm5, 16[eax]            ;(C1)   get third two 24-bit pixels
13      por             mm3, mm2        ;(B6)   combine the red, green, and blue
14      psrld           mm3, 6                  ;(B7)   shift to final position
15      por             mm1, mm0        ;(A6)   combine the red, green, and blue
16      movq            mm0, mm4        ;(D2)   save the original data
17      psrld           mm1, 6                  ;(A7)   shift to final position
18      pand            mm0, REDBLUE            ;(D3)   mask for 5MSbits of red and blue
19      packssdw mm1, mm3         ;(AB8)  pack result into one qword
20      movq            mm3, mm5        ;(C2)   save the original data
21      pmaddwd         mm0, mm7        ;(D4)   use multiply to shift and or
                                                ;       can't use mm0 for 3 cycles!
22      pand            mm3, REDBLUE            ;(C3)   mask for 5MSbits of red and blue
23      pand            mm4, mm6        ;(D5)   mask for 5MSbits of green
24      pmaddwd         mm3, mm7        ;(C4)   use multiply to shift and or
25      por             mm4, mm0        ;(D6)   combine the red, green, and blue
                                                ;       pipeline stall waiting for mm0
                                                ;       because pmadd needs 3 cycles
26      pand            mm5, mm6        ;(C5)   mask for 5MSbits of green
27      psrld           mm4, 6                  ;(D7)   shift to final position
28      por             mm5, mm3        ;(C6)   combine the red, green, and blue
                                                ;       pipeline stall waiting for mm3
                                                ;       because pmadd needs 3 cycles!
29      psrld           mm5, 6                  ;(C7)   shift to final position
30      packssdw mm5, mm4         ;(C8)   pack result into one qword
```

In order to achieve better pairing and make use of the three clock cycles after a PMADD instruction, the code can be adjusted to loop through the source code starting in the middle of the code shown in Example 5. By doing this, the instructions to read and begin processing the first two pairs of pixels can be done while waiting for the PMADD results in lines 21 and 24 to be usable and paired with lines 28 through 30. The loop control can also be added for "free" reducing the processing time to 17 cycles for eight pixels or 2.125 cycles per pixel. All of the instructions within the loop now pair. Of course, additional instructions are needed before the start of the loop in order correctly begin the loop. For large sets of data, the increase in efficiency quickly makes up for the extra cycles taken outside of the loop.

See Appendix B for the full source code for the PMADD algorithm. The instructions to calculate the value of the loop counter were written for this application and may need to be changed to meet the needs the user's application. Again, note that the source data is processed from the end of the source array instead of the beginning in order to eliminate the need for a compare instruction before the jump to the beginning of the loop.

APPENDIX A: Mask-Shift-Or Algorithm

```
;*
;* Description:
;*       The purpose of this file is to provide the MMX code for the
;*       RGB24to16 algorithm as an instructional example to those who
;*       are just beginning to code using MMX instructions.
;*
;* Assumptions:
;*       1.      The number of elements allocated for the source (src) must be
;*               divisible by 8. The number of rows X the number of columns does
;*               not need to be divisible by 8.  This is to allow working on 8
;*               pixels within the inner loop without having to post-process pixels
;*               after the loop.
;*       2.      The number of elements allocated for the destination (dest) must
;*               be divisible by 8.  The number of rows X the number of columns
;*               does not need to be divisible by 8.  This is to allow working on
;*               8 pixels within the inner loop without having to post-process
;*               pixels after the loop.
;*
;*************************************************************************/
;
.586
.MODEL FLAT, C
PD EQU <DWORD PTR>
PW EQU <WORD PTR>
PB EQU <BYTE PTR>
;----------------------------- -----------------------------------------
.CODE
RGB24to16 PROC C PUBLIC USES esi edi ebp ebx ecx, src:PTR DWORD,
                                                  dest:PTR DWORD,
                                                  nRows:PTR DWORD,
                                                  nCols:PTR DWORD
; Locals (on local stack frame)
saveesp                 EQU PD [esp+0]
EndOfLocals             EQU PD [esp+4]
LocalFrameSize = 4
;constants for MMX register initialization
.DATA
blues   dq 0f8000000f8h             ;mask for 5 MSbits of blue data
greens  dq 0f8000000f800h           ;mask for 5 MSbits of green data
reds    dq 0f8000000f80000h         ;mask for 5 MSbits of red data
.CODE
MEM_MASK_BLUES EQU DWORD PTR blues
MEM_MASK_GREENS EQU DWORD PTR greens
MEM_MASK_REDS EQU DWORD PTR reds
;************************************************************************
;*
;* Procedure: rgb24to16                                  Date: 12/08/95
;*
;* Author: Patricia L. Gray                              File: rgb24to16.asm
;*
;* Description:
;*       rgb24to16 is an optimized MMX routine to convert RGB data from
;*       24 bit true color to 16 bit high color.  The inner loop processes
;*       8 pixels at a time and packs the 8 pixels represented as 8 DWORDs
;*       into 8 WORDs. The algorithm used for each 2 pixels is as follows:
;*       Step 1:          read in 2 pixels as a quad word
;*       Step 2:          make a copy of the two pixels
;*       Step 3:          AND the 2 pixels with 0x00f8000000f8 to obtain a
;*                                quad word of:
```

```
;*      0000000000000000000000000BBBBB000 0000000000000000000000000bbbbb000
;*
;*      Step 4:          PSRLD quad word by 3 to obtain a quad word of:
;*      0000000000000000000000000BBBBB 0000000000000000000000000bbbbb
;*
;*      Step 5:          make a copy of the original again
;*      Step 6:          AND the copy of the original pixels with
;*                               0x0000f8000000f800 to obtain
;*      0000000000000000GGGGG00000000000 0000000000000000ggggg00000000000
;*
;*      Step 7           PSRLD quad word by 6 to obtain a quad word of:
;*      0000000000000000000000GGGGG00000 0000000000000000000000ggggg00000
;*
;*      Step 8           OR the results of Step 4 and 7 to obtain a quad
;*                               word of:
;*      0000000000000000000000GGGGGBBBBB 0000000000000000000000gggggbbbbb
;*
;*      Step 9           AND the last copy of the original with
;*                               0x00f8000000f80000h to obtain
;*      00000000RRRRR000000000000000000000 00000000rrrrr000000000000000000000
;*
;*      Step 10          PSRLD quad word by 9 to obtain a quad word of:
;*      0000000000000000RRRRR0000000000 0000000000000000rrrrr0000000000
;*
;*      Step 11          OR the results of Step 8 and 10 to obtain a quad
;*                               word of:
;*      0000000000000000RRRRRGGGGGBBBBB 0000000000000000rrrrrgggggbbbbb
;*
;*      Step 8:          When two pairs of pixels are converted, pack the
;*                               results into one register and then store them into
;*                               the destination.
;*
;* Inputs:
;*      src              long int *       a pointer to the first element
;*                                        of the input source
;*      dest             short int *      a pointer to the destination
;*                                        of the converted RGB data
;*      nRows            short int *      the number of rows in the src/dest bitmap
;*      nCols            short int *      the number of columns in the src/dest bitmap
;*
  mov    ecx,esp
  sub esp,LocalFrameSize
  and    esp, 0fffffff8h          ;8-byte align start of local stack frame
  mov    saveesp, ecx             ;save original esp to restore in epilgue
  mov    eax, nRows               ;multiply nRows
  mov    ebx, nCols               ;with nCols
  imul   ebx                      ;to get the size of the source
  mov    ecx, eax                 ;compare with loop counter
  sub    ecx, 8

  mov    eax, src                 ;load the src and
  mov    ebx, dest                ;and dest pointers
inner_loop:
  movq   mm0, [eax+4*ecx]         ;get the first 2 RGB elements

  movq   mm3, [eax+4*ecx+16]      ;get the third 2 RGB elements
  movq   mm1, mm0                 ;save original first 2

  pand   mm0, MEM_MASK_BLUES      ;mask out all but the 5 MSB blue data
  movq   mm4, mm3                 ;save the original third 2

  pand   mm3, MEM_MASK_BLUES      ;mask out all but the 5 MSB blue data
  psrld mm0, 3           ;shift blues right 3 which is now the result reg 1
```

```
  psrld mm3, 3                    ;shift blues right 3 which is now the result reg 3
  movq  mm2, mm1                  ;save the original again

  pand  mm1, MEM_MASK_GREENS      ;mask out all but the 5MSB green data
  movq  mm6, mm4          ;save the original again

  pand  mm4, MEM_MASK_GREENS      ;mask out all but the 5MSB green data
  psrld mm1, 6                    ;shift greens to bits 5-9

  por   mm0, mm1                  ;combine with the blue data in result reg 1
  psrld mm4, 6                    ;shift greens to bits 5-9

  pand  mm2, MEM_MASK_REDS        ;mask out all but the 5MSB red data
  por   mm3, mm4          ;combine with the green data in result reg 2

  pand  mm6, MEM_MASK_REDS        ;mask out all but the 5MSB red data
  psrld mm2, 9                    ;shift reds to bits 10-14

  por   mm0, mm2                  ;combine with the blue and green data in result reg
  psrld mm6, 9                    ;shift reds to bits 10-14

  movq  mm1, [eax+4*ecx+24]       ;get fourth 2 RGB elements
  por   mm6, mm3          ;combine with the blue and green data in result reg

  movq  mm3, [eax+4*ecx+8]        ;get second 2 RGB elements
  movq  mm2, mm1                  ;save the original fourth 2
  pand  mm1, MEM_MASK_BLUES       ;mask out all but the 5 MSB blue data
  movq  mm4, mm3                  ;save original second 2
  pand  mm3, MEM_MASK_BLUES       ;mask out all but the 5 MSB blue data
  psrld mm1, 3            ;shift blues right 3 which is now the result reg 4

  psrld mm3, 3                    ;shift blues right 3 which is now the result reg 2
  movq  mm5, mm4                  ;save the original again
  pand  mm4, MEM_MASK_GREENS      ;mask out all but the 5 MSB green data
  movq  mm7, mm2          ;save the original again
  pand  mm2, MEM_MASK_GREENS      ;mask out all but the 5 MSB green data
  psrld mm4, 6                    ;shift greens to bits 5-9
  por   mm3, mm4                  ;combine with the blue data in result reg 2
  psrld mm2, 6                    ;shift greens to bits 5-9
  pand  mm5, MEM_MASK_REDS        ;mask out all but the 5 MSB red data
  por   mm1, mm2                  ;combine with the blue data in result reg 4
  pand  mm7, MEM_MASK_REDS        ;mask out all but the 5 MSB red data
  psrld mm5, 9                    ;shift reds to bits 10-14
  por   mm3, mm5          ;combine with the blue and green data in result reg 2
  psrld mm7, 9                    ;shift reds to bits 10-14
  packssdw  mm0, mm3              ;pack result 1 and 2 into one qword
  por   mm7, mm1                  ;combine with the blue and green data in result reg 4
  packssdw  mm6, mm7              ;pack result 3 and 4 into one qword

  movq  [ebx+2*ecx], mm0          ;store the result

  movq  [ebx+2*ecx+8], mm6        ;store the result

  sub   ecx, 8
  jae   inner_loop        ;go get some more if not done

DONE:
  emms                            ; function epilogue
  mov    esp, saveesp
  ret
RGB24to16 ENDP
END
```

APPENDIX B: PMADD Algorithm

```
;* Description:
;*       The purpose of this file is to provide the MMX code for the
;*       RGB24to16 algorithm as an instructional example to those who
;*       are just beginning to code using MMX instructions.
;*
;* Assumptions:
;*       1.      The number of elements allocated for the pMatrix must be
;*               divisible by 8. The number of rows X the number of columns does
;*               not need to be divisible by 8.  This is to allow working on 8
;*               pixels within the inner loop without having to post-process pixels
;*               after the loop.
;*       2.      The number of elements allocated for qMatrix must be divisible
;*               by 8.  The number of rows X the number of columns does not need
;*               to be divisible by 8.  This is to allow working on 8 pixels within
;*               the inner loop without having to post-process pixels after the loop.
;*
;*****************************************************************************/
        TITLE   Convert RGB 24 to 16
        .486P
.model FLAT
;*****************************************************************************
;*                               DATA SEGMENT
;*****************************************************************************
_DATA   SEGMENT
rgbMulFactor            dq  2000000820000008H    ; RGB quad word multiplier
rgbMask1                dq  00f800f800f800f8H
rgbMask2                dq  0000f8000000f800H
_DATA ENDS
;*****************************************************************************
;*                               TEXT SEGMENT
;*****************************************************************************
_TEXT   SEGMENT
;
; Declare rgb24to16 as a public routine to allow the 'C' code to
; call it.
;
PUBLIC RGB24to16
;* Description:
;*       rgb24to16 is an optimized MMX routine to convert RGB data from
;*       24 bit true color to 16 bit high color.  The inner loop processes 8
;*       pixels at a time and packs the 8 pixels represented as 8 DWORDs
;*       into 8 WORDs. The algorithm used for each 2 pixels is as follows:
;*       Step 1:          read in 2 pixels as a quad word
;*       Step 2:          make a copy of the two pixels
;*       Step 3:          AND the 2 pixels with 0x00f800f800f800f8 to obtain a
;*                               quad word of:
;*       00000000RRRRR000000000000BBBBB000 00000000rrrrr000000000000bbbbb000
;*
;*       Step 4:          PMADDWD this quad word by 0x2000000820000008 to obtain
;*                               a quad word of:
;*       000000000000RRRRR00000BBBBB000000 000000000000rrrrr00000bbbbb000000
;*
;*       Step 5:          AND the copy of the original pixels with
;*                               0x0000f8000000f800 to obtain
;*       0000000000000000GGGGG00000000000 0000000000000000ggggg00000000000
;*
;*       Step 6:          OR the results of step 4 and step 5 to obtain
;*       000000000000RRRRRGGGGGBBBBB000000 000000000000rrrrrgggggbbbbb000000
;*
```

```
;*      Step 7:          SHIFT RIGHT by 6 bits to obtain
;*      0000000000000000RRRRRGGGGGBBBBB 0000000000000000rrrrrggggggbbbbb
;*
;*      Step 8:          When two pairs of pixels are converted, pack the
;*                               results into one register and then store them into
;*                               the q Matrix.
;*
;* Inputs:
;*      pPtr             long int  *      a pointer to the first element
;*                                        of the input 'p' matrix
;*      qPtr             short int *      a pointer to the output
;*                                        RGB converted matrix
;*      nRows            short int *      the number of rows in the p/q matrix
;*      nCols            short int *      the number of columns in the p/q matrix
;*
;*****************************************************************************/
RGB24to16 PROC C USES ebx ecx edx,
pMatrix:PTR DWORD,
qMatrix:PTR WORD,
nRows:DWORD,
nCols:DWORD
;
; This calculates the number of elements in the 'p' matrix and assigns it to
; EAX.  EAX is then adjusted to contain the index to the last 8 pixel aligned
; pixel by performing ((nRows * nCols) - 8 + 7) & 0xfffffff8. Pointers to the
; arrays 'p' and 'q' are also set up
;
        mov     eax, nRows
        mov     ebx, nCols

        imul    ebx                             ; EAX = total number of
                                                ; pixels to process
        mov     ebx, pMatrix            ; EBX points to 'pMatrix'
        sub     eax, 1                  ; align index EAX to the
                                                ; last 8 pixel boundary
        mov     edx, qMatrix            ; EDX points to 'qMatrix'
        and     eax, 0fffffff8H         ; finish the index EAX
                                                ; alignment
;
; This section performs up to and including step 4 on pixels 0 and 1.  It also
; performs up to and including step 5 on pixels 2 and 3.  This is done prior to
; entering the loop so that better loop efficiency is achieved.  Better loop
; efficiency is achieved because these instructions are paired with other
; instruction at the end of the loop which could not be previously paired.
;
        movq    mm7, DWORD PTR rgbMulFactor     ; MM7 = pixel multiplication
                                                        ;factor
        movq    mm6, DWORD PTR rgbMask2         ; MM6 = green pixel mask
        movq    mm2, 8[ebx][eax*4]              ; get pixels 2 and 3
        movq    mm0, [ebx][eax*4]                       ; get pixels 0 and 1
        movq    mm3, mm2                        ; copy pixels 2 and 3
        pand    mm3, DWORD PTR rgbMask1         ; get R and B of pixels 2 and 3
        movq    mm1, mm0                        ; copy pixels 0 and 1
        pand    mm1, DWORD PTR rgbMask1         ; get R and B of pixels 0 and 1
        pmaddwd mm3, mm7                        ; SHIFT-OR pixels 2 and 3
        pmaddwd mm1, mm7                        ; SHIFT-OR pixels 0 and 1
        pand    mm2, mm6                        ; get G of pixels 2 and 3
;
; This section performs steps 1 through 8 for 4 pairs of pixels (or for a total
; of 8 pixels).
;
inner_loop:
        movq    mm4, 24[ebx][eax*4]     ; get pixels 6 and 7
        pand    mm0, mm6                ; get G of pixels 0 and 1
```

```
        movq    mm5, 16[ebx][eax*4]     ; get pixels 4 and 5
        por     mm3, mm2                ; OR to get RGB of pixels 2
                                                ; and 3
        psrld   mm3, 6                  ; SHIFT pixels 2 and 3 (step 7)
        por     mm1, mm0                ; OR to get RGB of pixels 0
                                                ; and 1

        movq    mm0, mm4                ; copy pixels 6 and 7
        psrld   mm1, 6                  ; SHIFT pixels 0 and 1 (step 7)
        pand    mm0, DWORD PTR rgbMask1 ; get R and B of pixels 6 and 7
        packssdw mm1, mm3                       ; combine pixels 0, 1, 2, and 3
        movq  mm3, mm5                  ; copy pixels 4 and 5
        pmaddwd mm0, mm7                ; SHIFT-OR pixels 6 and 7
        pand  mm3, DWORD PTR rgbMask1   ; get R and B of pixels 4 and 5
        pand  mm4, mm6                  ; get G of pixels 6 and 7

        movq    [edx][eax*2], mm1               ; store pixels 0, 1, 2, and 3
        pmaddwd mm3, mm7                ; SHIFT-OR pixels 4 and 5
        sub     eax, 8                  ; subtract 8 pixels from the index
        por     mm4, mm0                ; OR to get RGB of pixels 6 and 7

        pand    mm5, mm6                ; get G of pixels 4 and 5
        psrld   mm4, 6                  ; loop iteration; SHIFT pixels 6
        movq    mm2, 8[ebx][eax*4]      ; get pixels 2 and 3 for the next
        por     mm5, mm3                ; loop iteration ; OR to get RGB of
                                                ; and 7 (step 7)
        movq    mm0, [ebx][eax*4]               ; get pixels 0 and 1 for the next
        psrld   mm5, 6                  ; SHIFT pixels 4 and 5 (step 7)
                                                ; pixels 4 and 5
        movq    mm3, mm2                ; copy pixels 2 and 3
        movq    mm1, mm0                ; copy pixels 0 and 1
        pand    mm3, DWORD PTR rgbMask1 ; get R and B of pixels 2 and 3
        packssdw mm5, mm4                       ; combine pixels 4, 5, 6 and 7

        pand    mm1, DWORD PTR rgbMask1 ; get R and B of pixels 0 and 1
        pand    mm2, mm6                ; get G of pixels 2 and 3
        movq    24[edx][eax*2], mm5     ; store pixels 4, 5, 6 and 7
        pmaddwd mm3, mm7                ; SHIFT-OR pixels 2 and 3

        pmaddwd mm1, mm7                ; SHIFT-OR pixels 0 and 1
        jge     inner_loop                      ; if we need to do more jump to the
                                                ; beginning of the loop
;
; We have converted 24-bit true color to 16-bit high color for the given data!
;
rgb24to16_done:
        emms
        ret     0                               ; we are done!
RGB24to16 ENDP
_TEXT   ENDS
END
```